WiGis: A Framework for Scalable Web-based Interactive Graph Visualizations

Brynjar Gretarsson* Svetlin Bostandjiev† John O'Donovan‡
Tobias Höllerer§
Department of Computer Science
University of California, Santa Barbara.

Abstract

Traditional network visualization tools inherently suffer from scalability problems, particularly when such tools are interactive and web-based. In this paper we introduce *WiGis* –Web-based Interactive Graph Visualizations. *WiGis*[1] exemplify a fully web-based framework for visualizing large-scale graphs natively in a user's browser at interactive frame rates with no discernible associated startup costs. We demonstrate fast, interactive graph animations for up to hundreds of thousands of nodes in a browser through the use of asynchronous data and image transfer. Empirical evaluations show that our system outperforms traditional web-based graph visualization tools by at least an order of magnitude in terms of scalability, while maintaining fast, high-quality interaction.

1 Introduction

This paper presents a novel visualization framework which supports user interactions with large graphs in a web browser without the need for plug-ins or special-purpose runtime systems. Our framework follows the common visual information browsing principle: "Overview first, zoom and filter, then details on demand"[15] for exploration of large information spaces. The *WiGis* framework supports information discovery in two phases. Firstly, by enabling users to visualize and interact with large scale network data, our framework provides a "big picture" of the information space. Secondly, interaction is used to mold large scale data into the user's own mental model, which serves as a useful starting point for more fine-grained analysis.

Many tools for visualizing graphs have previously been developed. Some of these tools run in a web-browser [6, 16, 17] while others are scalable for up to hundreds of thousands of nodes [2, 14]. However, to our knowledge, no web-based tools exist which are capable of interactive visualization of graphs at such scale [13]. We have developed an extensible framework which enables interactive visualization of hundreds of thousands of nodes natively in a web browser.

Based on our analysis of existing interactive web-based graph visualization systems, we find that their main scalability limitation is due to the fact that most of them implement some form of a thick client solution and subsequently need to load the entire graph onto the client machine. In addition to large startup costs, this limits the maximum size of visualized graphs due to memory limitations of the browser or the client computer. We circumvent these limitations by leveraging a novel technique for storing and processing graph data on a powerful remote server. The server continuously produces bitmap images and asynchronously sends them to the client's browser. This provides a smooth interactive animation natively in the browser. For example, we achieve more than 10 frames per second for graphs of 10,000 nodes and 20,000 edges, with minimal requirements for memory and processing power on the client machine. We believe that this is an important contribution which supports interactive visualization of large graphs even on machines with limited resources, such as mobile devices.

We use the term *WiGis*, or Web-based Interactive Graph Visualizations, for our framework, which is a new addition to the set of currently available tools providing "visualization as a service" [6]. This paper describes

*brynjar@cs.ucsb.edu
†alex@cs.ucsb.edu
‡jod@cs.ucsb.edu
§holl@cs.ucsb.edu
[1] www.wigis.net

and evaluates the new framework with a range of test datasets with focus on performance in terms of speed and scalability. Furthermore, we present a comparative experiment in which our system exhibits similar results to the best performing desktop applications, while supporting visualization of over one million nodes, albeit at lower frame rates. None of the tested systems could visualize graphs of this size. In comparison with other web-based systems, *WiGis* improved on the next best performer by an order of magnitude in terms of scale while also performing at least as well or better in terms of speed.

The remainder of this paper is organized as follows: A critical review of current relevant work in the area of large graph visualizations with a focus on web-based approaches is presented in Section 2. Section 3 describes the architecture of *WiGis* in terms of design and implementation. Section 4 discusses an empirical evaluation of our visualization tool (and its component parts) in terms of scalability and speed with respect to popular graph visualization tools. Section 5 contains a brief discussion of the benefits and limitations of our technique as well as various deployments of the system. The paper concludes with a summary of the main contributions.

2 Background

Much research has been conducted on large scale graph visualizations, e.g. [2, 8, 10, 14]. Traditionally, graph visualization applications have been desktop based. For example, Cytoscape [14], Pajek [4], Tulip [2], and some implementations of Tom Sawyer Visualization [16]. Over the past few years, increased web-accessibility and bandwidth improvements have triggered a general shift towards rich internet applications (RIAs) capable of providing interactive and responsive interfaces through a web browser. This shift has a potential benefit for resource-intensive graph visualization, and applications which take advantage of the rich-internet paradigm are beginning to emerge for visualization of graph and network data. Examples of such applications include Touchgraph [17], Tom Sawyer Visualization [16] and IBM's Many Eyes [6]. In general these applications either do not scale past thousands of nodes or are not fully interactive.

Thick v/s Thin Clients RIAs can be loosely classified into thick and thin clients. A thick client typically provides rich functionality that is largely independent of a central server with the majority of processing done on the client, whereas a thin client requires constant communication with a server to provide functionality. Client-based visualization [20, 1] can be considered a thick-client solution since data is downloaded from a server and the visualization and rendering are done at the client side. The popular graph visualization tool Touchgraph Navigator [17] is a good example of a client-based tool, since it processes graph interactions locally in Java. A thick client solution needs to initially download the entire graph data, which may be on the order of GBs for large graphs, onto the client machine. This severely limits the size of the largest graph a thick client application can handle and poses significant startup times. Once the graph data is obtained there is no guarantee that the client has enough memory available to handle the data. Visualizations can be created on a remote server and passed across a network to the client. This is referred to by Wood et al. [20] as "server-based" visualization, and is an example of a thin client solution. *WiGis* uses both client-based and server-based visualization techniques. An important innovation is the way the system can automatically and transparently switch between the two modes while allowing smooth interaction in both.

Plug-in v/s Native Applications RIAs can be further classified based on the manner in which they are deployed. Many RIAs are implemented using some form of browser plug-in, for example Java Applets, Adobe Flash or Microsoft Silverlight. The majority of graph visualization tools available on the web are plug-in based, e.g. [17, 6]. There are some fundamental drawbacks with the plug-in approach however. Firstly from a scalability perspective, plug-in based RIAs are limited to the capabilities of the plug-in itself. For example, the default memory limit for Java Applets is usually around 60-90 MB. Furthermore, from a security perspective third party plug-ins are usually not open source and need to access client resources, making them a potential security threat. Many large organizations with sensitive data require every line of third party code to be checked for malicious behaviour before deployment on an analysts machine. The

alternative approach to plug-in based RIA implementation is to provide functionality natively in the browser through a combination of DHTML and AJAX. Since this approach does not require any access to client resources outside of the browser it is much more secure with respect to integrity of the client machine. Examples of native RIA's include Google Maps and the JSP and ASP.net implementations of Tom Saywer Visualization [16]. Since we are concerned about scalability we opted to design our framework as a native RIA to avoid the inherent limitations and security drawbacks of browser plug-ins.

While plug-ins such as Flash and Applets provide rich functionality such as object support and dynamic components, they are not directly suited to solve the problem of large scale graph visualization on the web. In order to fully utilize these rich features for visualization of large graphs the entire graph model would have to exist on the client machine. For large graphs this is not a feasible solution due to potential memory limitations on the client machine. The only feasible solution is to store the graph data on a server and pass bitmap images of the graph across the network. While such images can be displayed inside a plug-in, the visualization framework would not benefit from the rich functionality of the plug-in. A simpler approach is to display the image natively in a browser, thus eliminating other drawbacks of the plug-ins, such as memory limitations, start-up cost, and security issues.

3 Architecture

To reiterate, the main contribution of this work is a scalable web-based technique for providing smooth interaction with very large graphs in a user's web-browser. Figure 1 describes the novel, lightweight and flexible architecture we use to achieve this goal. This architecture does not rely heavily on a client's resources, and requires only a basic browser with no external plug-ins. Two modes of operation are supported: client-mode and server-mode. In server-mode the client's browser only contains a single bitmap image of the graph. All layout and interaction algorithms are run on the server and the bitmap image in the browser is updated on the fly, giving a very smooth interaction experience. In client-mode the browser represents each node as an image and each edge as an SVG line. Layout and interaction algorithms run locally in the browser. The server always maintains a model of the entire graph, while the browser can have anywhere from the whole graph model (client-mode with a small graph) down to no graph model at all (server-mode). The system automatically switches between these two modes of operation based on the size of the graph being displayed at a given time.

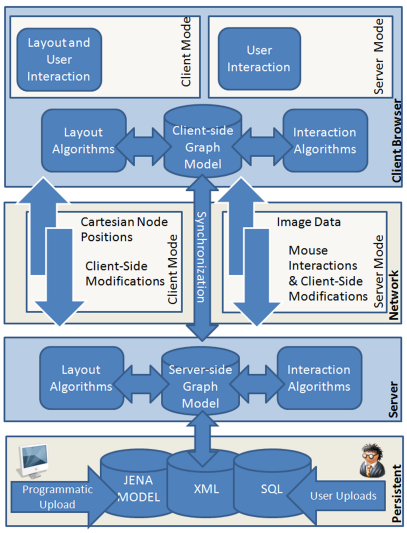


Figure 1: A scalable web-based architecture for interactive graph visualization, as used in *WiGis*

At the outset of this work, a primary concern was the refresh rate that could be achieved with this type of design. When considering rendering time, network delay, and other processing overheads, one is tempted to picture a slow, jumpy interaction experience. However, as we prove in our evaluations and in our online demonstration[2], this design does achieve fast, smooth graph interactions for up to hundreds of thousands of nodes, even when these potential bottlenecks are considered.

3.1 Visualization Modes

Client-Mode When a graph in the viewing window (shown in Figure 2) is sufficiently small, all layouts, interactions and renderings are performed in the client browser. This can be either the whole graph or a focused area of a larger graph. The top layer in Figure 1 represents the browser, which contains a model of the graph, referred to as the client-side model. As this model is updated by JavaScript layout and interactions, its

[2] www.wigis.net

state is asynchronously transferred to a remote server, which updates a server-side graph model accordingly (shown in the third layer from top in Figure 1). Rendering is performed by JavaScript using SVG for edges (replicable in VML for Internet Explorer) and HTML image tags for nodes. This combination was chosen because it exhibited the best performance out of a multitude of rendering options over a combination of metrics in our preliminary tests. In client-mode, *WiGis* can still make use of rich server-side functionalities such as clustering for instance. The client simply calls the remote function on the server through AJAX, the server runs a process, updates its model and passes it to the client.

The following list outlines the motivations for, and benefits of using client-side graph processing:

Very Smooth Interaction - Client side computations provide fast interactions for small graphs because there is no direct network overhead.

Network Independent - Client-side processing does not need a fast network to function well, and can even function in an off-line state.

Easy on Server Resources - With a potentially large user base, *WiGis* can be heavy on server resources. Utilizing client resources wherever possible eases the load on a centralized server or server set. We note that in client-mode, the remote server still holds a model of full the graph, so only CPU load is reduced, as opposed to memory.

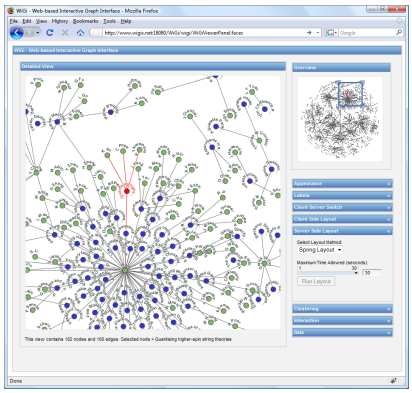


Figure 2: A screenshot of *WiGis* displaying results expanded from the seed query "Graph Visualization" on the Citeseer dataset. Graph shows 1104 author and article nodes, with 1125 associations. An overview window of the entire graph is shown in the top right corner with the detail view highlighted by the zoom box.

Server-Mode For large graphs, *WiGis* automatically switches into server-mode. In this mode, all computations for both layout and interaction are processed on the remote server. Instead of passing a graph model back to the client browser for reconstruction, the server generates a bitmap image of the updated graph. This image is passed across the network and rendered in the browser. Swapping from client to server mode is a *seamless transition* for the end-user, with no jumpiness or image differences, as shown in Figure 3. While in server-mode, interaction is facilitated by capturing mouse interactions on the image of the graph using JavaScript. Mouse interactions are passed asynchronously to the server and the interaction/layout algorithms are triggered on the server-side graph model based on the new input. The server computes an updated graph, renders it, and sends an image of the rendered graph back to the client. The key success of our tool lies in the fact that this entire process occurs at interactive speeds giving very smooth desktop-like interactions with very large graphs.

Our system achieves update rates of 10 frames per second for graphs up to the order of 10K nodes, while graphs of the order of 100K nodes are rendered at approximately 1 to 2 frames per second (c.f. results in Section 4.6). Theoretically, with sufficient hardware resources on the server-side, the upper bound for the number of nodes *WiGis* can usefully display in an interactive fashion approaches the pixel resolution of the client display.

Server-side operation of *WiGis* can loosely be compared to a Google-Maps interface with the difference that transmitted images are not static or pre-defined. Instead, images are computed on-the-fly based on a combination of user input and the existing graph state. The following list shows the benefits and drawbacks of using the server-side approach for large graph computation.

Scalability - Client side graph visualizations generally fail as the graph size approaches thousands of nodes and edges. Using our server-side technique we can interactively visualize graphs of up to 1 millon nodes natively in the browser.

Remote Resources - Server-side processing extends the power of the browser well beyond the resources of the local machine by using a thin client implementation.

Bandwidth Limitation - Server side graph processing relies heavily on network resources, and can perform poorly on slow networks. While many universities operate very fast connections, home and wireless

broadband connections typically range from 64 kb/s to about 1 Mb/s. Our evaluations show that network overhead for the image transfer becomes negligible for graphs of over 100 thousand nodes.

3.2 System Architecture Layers

Following is a description of the architecture based around the four layers in Figure 1 from top to bottom. These layers represent physical locations or communications between them, as opposed to the previously discussed client-mode and server-mode, which are modes of *operation* spanning across all layers, and are depicted by the vertical data flows in Figure 1.

Client Browser Layer The top layer in Figure 1 represents a web-browser running on a client machine. Depending on the mode of operation, the browser holds either a JavaScript model and an SVG/HTML visualization of the graph (client-mode) or a single bitmap image of the graph in its current state (server-mode). The browser contains a JavaScript implementation of a selected layout algorithm and a selected interaction algorithm, *both of which are scripted "replicas" of server-side algorithms.* Depending on the current operation mode (client or server), the browser layer communicates either graph model data or mouse interaction data across the network to the remote server.

Server Layer The server *layer* is the "powerhouse" of *WiGis*, where most of the heavy processing occurs for large graphs. The server holds a model of the full graph (in memory if possible), a set of graph layout, clustering, and interaction algorithms (currently implemented in Java, but extensible to any language). The key concept of the architecture is that the client layer mirrors the server graph model to the capacity of its available resource. Again, depending on the scale of the visible part of the graph and resources available on the client, the server either accepts mouse interaction data (server-mode) or an updated graph model (client-mode) from the client browser. In return, the server communicates either graph model data or GIF images back to the browser depending on the current mode of operation. The graph model on the server is always kept in synch with the client model through AJAX updates.

Network Layer The network layer in Figure 1 represents the communication between the server and client layer. Depending on the mode of operation, image data and interaction data (server-mode) or graph model data (client-mode) is sent across the network to maintain synchronization between the client and server layers.

Persistent Layer Graph data can be uploaded to the system through a web interface by users or programmatically by other systems to add interactive visualization capabilities to them. User uploads are provided in several common formats including XML, GraphML and a simple CSV representation. Regardless of the original source, all data is converted to an XML representation and read into the graph server. The persistent layer of *WiGis* is kept modular to allow data from a broad range of sources to be plugged in easily. For instance, current data sources include citation data from a publication search tool and dynamically generated topic models from New York Times articles.

3.3 Client/Server Implementations

Each algorithm in the client browser is coded in JavaScript to exactly mimic the corresponding server-side java version. The algorithms are designed to be identical with one exception: the client side algorithm operates only on a subgraph containing all visible nodes and their immediate neighbors. This constraint is necessary because of the scalability limitations of JavaScript and the potentially limited resources on the client machine. Since we must use different platforms and implementation languages, there are also small differences between the resultant graph layouts. However, in most cases these differences are too small to be discerned visually by the end-user. Figure 3 depicts a sample graph visualized in (a.) client-mode and (b.) server-mode. It is clear from Figure 3 that the two representations are very similar, although they might

be misaligned by one pixel as a result of floating point errors in the conversion between different coordinate systems. Other minor differences exist in the anti-aliasing of the lines and circles. The system automatically switches from server-mode to client-mode when zoomed into a sufficiently small part of the graph and back to server-mode when zoomed out to a larger portion of the graph.

3.4 Layout

The *WiGis* architecture is modular and provides an interface for plugging in multiple different layout algorithms which can then be selected through the user interface. The focus of this paper is on scalability with regard to interaction, and accordingly, details of various layout algorithms are not included here. For the purpose of our analysis we use an efficient implementation of a simple force-directed graph layout algorithm [7] [9].

(a) Client-Mode (b) Server-Mode (c) Difference

Figure 3: Seamless transition between client and server renderings. (a.) Client-Mode. Rendering and layout done with DHTML and SVG. (b.) Server-Mode. Rendering and layout done in Java on a remote machine. (c.) Difference image between a and b.

3.5 Interaction

Interaction with large graphs is not as straightforward as interaction with a small number of nodes, since moving one node at a time can be very time consuming when molding a layout of thousands of nodes. Moreover, the commonly used rectangular area selection of multiple nodes that happen to lie close to each other is not ideal when interacting with large graphs because the selected nodes do not necessarily have meaningful associations with each other apart from proximity as computed by a layout algorithm. For our framework we have chosen an interaction algorithm, which we refer to as the interpolation method, originally developed by Trethewey and Höllerer for use in a desktop application [18]. This method gives great performance for large graphs in terms of speed while also making it easy for users to mold a large graph.

4 Evaluation

Now that we have described our technique for enabling interactions with large graphs on the web, we focus on an empirical evaluation of the technique in terms of speed and scalability. To properly evaluate our system we break down the interactive visualization process into its component pieces and present evaluations of each individually, before testing the system as a whole. As a precursor to this, we define a test dataset of graphs at different scales and discuss their properties. All of the following experiments use the same test data. For the purpose of this evaluation we define three steps (potential bottlenecks) in our interactive visualization process:

Step 1: Rendering - Drawing graphs after a change has been made.

Step 2: Interaction - Capturing user input and calculating modifications to the graph.

Step 3: Network - Passing graph and/or image data across the network from a remote server.

After evaluating each step in isolation, we combine our results to produce our estimated overall time for the variety of graphs in our test suite. As a sanity-check this is then compared against recorded times for interaction with the system as a whole. A discussion of the relative impact of each step is presented. Our evaluation concludes with a comparison against three popular interactive graph visualization systems.

4.1 Setup

All experiments were performed on a 64 Bit Dell Inspiron 530 with an Intel Q9300 2.5GHz quad core processor, 8GB of RAM, an ATI Radeon HD 3650 video card and a serial ATA hard drive with 7200 rpm.

The operating system was Windows Vista SP1. No other heavy processes were allowed to run during experiments. A Dell 24" UltraSharp flatscreen monitor with a refresh rate of 60Hz was connected with a DVI cable. Screen resolution was constant at 1920x1200 pixels for all experiments. Graph window sizes were kept constant at 600x600 pixels. This value was chosen because it will fit in most browser windows with 1024x768 resolution. Frame rates were recorded either by *WiGis* or by FRAPS[3]. On our multi core machine, FRAPS did not introduce significant delays in any renderings. All web-based experiments were performed in Mozilla Firefox 3.0.3 and in Google Chrome 1.0.1 with no plug-ins or add-ons running. *WiGis* is Java-based and hosted on a JBoss 4.2.2 server running on the same machine as the client browser. This was done to control network overhead which allowed us to determine the theoretical network overhead of all connection speeds and provide a more reliable and consistent result set based on the the exact size of the data which is passed across the network. It should be noted that we have also successfully tested the system across a real network, with similar results. Moreover, *WiGis* is currently accessible on the world wide web at `www.wigis.net`.

4.2 Description of Test Data

There are many possible approaches for testing a web-based system for large-graph layout and interaction. Our system works well with real world data, for example citation networks, computational provenance graphs and topical relations among newspaper articles. We have also applied our system successfully to specific graph types such as meshes, trees, highly connected and highly sparse data. For this paper we have chosen to perform our tests on "small world" data [3, 19], because it is abundant in the social web, financial, biological and many other naturally occurring networks [12]. Small world networks are connected graphs in which most nodes are not direct neighbors but can be reached in a small number of hops from most points in the graph. Our test data was generated using the Barabási-Albert (BA) Model [3] for creation of small world networks. The BA model uses preferential attachment [3] for the addition of new nodes. Table 1 describes our test graphs $G1, ..., G6$. Graph size is increased exponentially from $G1$ (10 nodes, 20 edges) to $G6$ (1M nodes, 2M edges). To confirm the small world nature of our test data, the degree of connectivity versus number of nodes for all graphs was plotted on a log-log scale. This test produced linear trends in logarithmic space for all graphs, exhibiting the trademark power-law distribution of small world networks [3]. Our test data and graph analysis are available for download online[4].

Graph	G1	G2	G3	G4	G5	G6
Nodes	10	100	1K	10K	100K	1M
Edges	20	200	2K	20K	200K	2M

Table 1: Description of generated small-world data

4.3 Rendering

This section describes the procedure and results of an experiment to test the first of three potential bottlenecks for our system. This bottleneck occurs while re-rendering the graph after it has been modified. Modifications can happen either on client or server side, directly by the user, or by interaction and layout algorithms. To evaluate rendering speeds in both client and server modes, each of our test graphs $G1, ..., G6$ was rendered and the average time per frame was recorded. Graphs were redrawn at every frame and all nodes and edges were constrained to the viewing window for the entire test.

As described in Section 3.1, client-side rendering was performed by JavaScript using SVG for edges and HTML image tags for nodes. In server-mode, graphs were rendered into GIF images using Sun Microsystems Java2D graphics library from Java 6.0 and those images were passed to the browser. Our standard setup (outlined in Section 4.1) was applied. For this test, edge width was kept constant at 1 pixel and node size was a constant 4x4 pixels.

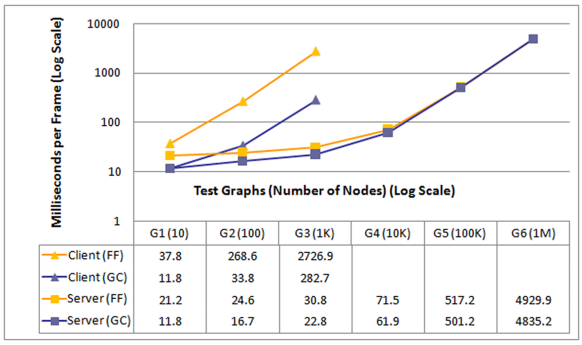
Figure 4 shows the results of the rendering experiment. Since our test graphs increase exponentially in size, results are presented on a log-log scale. Note that on this scale, *small differences at upper parts of the graphs represent significantly larger differences on a linear scale*. The four plots in Figure 4 represent

[3] www.fraps.com

[4] www.wigis.net

the client and server side frame rates in ms/frame for rendering of test graphs $G1, ..., G6$ in both Mozilla Firefox (FF) and Google Chrome (GC). The graph shows that both client and server methods eventually scale approximately linearly with number of nodes, however the client side method is notably slower than the server side, because the browser takes longer to update position data in the document object model as graph size is increased. The smaller graphs create a curve effect because of various overheads, but importantly, the larger graphs $G4, ..., G6$ show a linear trend.

In fact, the server side method performs better than linear, as the overhead of loading the image into the browser and displaying it is almost constant for all graph sizes. At $G1$ the methods have identical performance, while at $G3$ the difference is 260 ms for GC and approaches 3 seconds in FF. For small graph sizes, typically less than 100 nodes, client side rendering takes about the same time as server side rendering, in the range of 0 to 260 ms/frame. The client-side approach could not scale past test graph $G3$ (1K nodes). This occurs either because the max number of SVG lines the browser is capable of rendering is exceeded and the page fails to load (FF), or because the browser slows to an unusable state (GC). Irrespective of these failures however, there is stronger motivation for using the server side method for large graphs: at about 1000 nodes it simply becomes more efficient to pass an image of the rendered graph across the network as opposed to sending raw node/edge data and rendering it on the client.



	G1 (10)	G2 (100)	G3 (1K)	G4 (10K)	G5 (100K)	G6 (1M)
Client (FF)	37.8	268.6	2726.9			
Client (GC)	11.8	33.8	282.7			
Server (FF)	21.2	24.6	30.8	71.5	517.2	4929.9
Server (GC)	11.8	16.7	22.8	61.9	501.2	4835.2

Figure 4: Results of the rendering experiment showing milliseconds per frame at various graph sizes.

Client side rendering consists of two parts: drawing of SVG lines for edges and HTML image tags for nodes. In addition to the results in Figure 4, these two parts were tested individually. Averaged across all test graphs, line drawing took 53% and image repositioning took 47% of the total rendering time.

Clear performance differences were exhibited between browsers, Chrome was significantly faster than Firefox, which took an average of 6 and 1.3 times longer for client and server methods respectively across all test graphs. This result is as expected because more work is being done by the browser while in client-mode, and Google Chrome's V8 JavaScript engine is faster than the Firefox 3.0.X engine. Looking forward, improvements to JavaScript engines are underway in most major browsers, and we expect that our technique will perform better as faster engines are released.

This experiment shows that our technique is capable of rendering graphs in a browser in the order of hundreds of thousands of nodes and edges in a fraction of a second. This is clearly indicated by the results in Figure 4, which show that $G5$ (100K nodes) takes about half a second while $G6$ (1M nodes) takes less than 5 seconds. It is important to note that although 5 seconds may not seem fast, it is a significant improvement over the other systems which could not load $G6$. Also, our experiment represents a worst-case scenario where all geometric primitives are redrawn. This happens only when the user is working at the overview level that contains the entire graph. If the user is in a zoomed-in view where a smaller number of primitives need to be redrawn for each frame, rendering may be on the order of milliseconds.

4.4 Interaction

The second potential bottleneck in our framework is the simple but effective interaction algorithm we used. However, after evaluation it became clear that the relative time spent on the interaction algorithm was very little as shown in Table 3. For this reason, and due to space restrictions this evaluation is not discussed here.

4.5 Network

Graph	$G1$	$G2$	$G3$	$G4$	$G5$	$G6$
Avg. Img Size	6.5	20.5	34	32	34	29

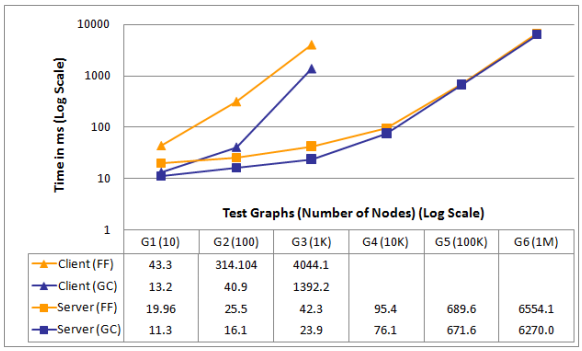
Table 2: Average image size in kB for each test graph

The third and final factor in the component analysis of our technique is a look at various network delays that occur as we pass data asynchronously between the client and server. In client-mode, network delay is minimal since we are only passing initial layout data for graphs of the order of $G3$ or less. Additionally, rendering and interactions are computed locally, so there is no network delay for interaction. However, updates from the client model are passed across the network to maintain synchronicity between client and server graph models. This allows us to switch to server-mode at any time. In server-mode, network capacity has a severe impact on system performance since images of rendered graphs are constantly passed from server to client side. Table 2 shows the average image size in kB for all graphs in our test set. Assuming a network speed of 1000 kB/s (which is common for university campuses) the values in Table 2 are also equivalent to the transfer time in milliseconds for each image. Table 3 presents a breakdown of the total interactive visualization process with network delays included. For graphs of about 1 million nodes network delay represents less than 1% of the total processing time. For smaller graphs, e.g $G3$, the delay can account for about 59% of the entire process since the size of an image of the rendered graph is relatively stable across graphs $G3, ..., G6$. We also evaluated the amount of delay introduced by a slower connection of 1000 kb/s which is a common connection speed for residential areas in the USA. This would obviously introduce eight times more network overhead, resulting in about 330 ms per frame for $G4$ and about 890 ms per frame for $G5$. Interaction with $G6$ would still be under 6 seconds per frame. Again, we note that no other system tested was able to load graphs of the order of $G6$.

4.6 Putting It All Together

Up to this point, we have focused on analysis of the various components of *WiGis* at an individual level. Now we put them all together to evaluate the performance of the system as a whole. This evaluation is performed in two parts, firstly an analysis of speed and scalability is presented based on our test data. Secondly, we present a comparison of our technique against three popular graph visualization systems with respect to speed and scalability.

Scalability Test Figure 5 shows the time in milliseconds for the full interactive visualization process on Graphs $G1, ..., G6$, which includes rendering, interaction and network delays. These results are for interaction with the entire graph, i.e: the effect parameter was set to maximum value, making interactions effect every node. This represents the worst case scenario for our system since every node is repositioned in each frame. The test was run in Firefox (FF) and Google Chrome (GC) browsers in client and server modes. There is an obvious difference in scalability between client and server modes. At $G1$ (10 nodes) there is only a few milliseconds difference between them, but at $G3$ (1000 nodes) the client process is taking 96 times longer than the server (4044 ms compared with 42.3 ms). Again in this test we can see that for the client side process, FF is far slower than GC, taking 4.6 times longer on average. The surprising result in this test is that our technique for computing graphs remotely (i.e: the server side method) is actually faster than JavaScript for large *and* small graphs. (in GC, 1.2 times faster for $G1$, 2.5 times faster for $G2$, and 58 times faster for $G3$). The test was also performed with single node interaction and a similar trend was revealed. For full graph interaction in GC, the million node graph ($G6$) took about 6.3 s, while the single node interaction took 5.7 s. It is important to notice that the rendering of $G6$ took 77% of the total time and, as noted earlier, if the user is working at a zoomed-in level instead of at the overview level the rendering time may be significantly smaller. Thus,



	G1 (10)	G2 (100)	G3 (1K)	G4 (10K)	G5 (100K)	G6 (1M)
Client (FF)	43.3	314.104	4044.1			
Client (GC)	13.2	40.9	1392.2			
Server (FF)	19.96	25.5	42.3	95.4	689.6	6554.1
Server (GC)	11.3	16.1	23.9	76.1	671.6	6270.0

Figure 5: Results of interactive visualization experiment, showing average times per frame for the worst case scenario, where every node is repositioned in every frame.

interaction with the 1 million node graph may take as little as 1 second per frame.

Delay Breakdown To gain an understanding of the delays caused by each part of the online interactive visualization process we computed a percentage analysis for each step over all of our test graphs. Table 3 outlines the results for graphs $G1, ..., G6$. For $G1$ and $G2$, client-mode was used because this is the system default for small graphs and gives the best performance in most cases. The table shows the percentage time for rendering, interaction, and the expected network costs. The total column shows an empirically tested value for the entire process over each graph. The difference between the total and the sum of component pieces is shown as "Other". We suspect that this value is due to various system processes, browser overheads, other unmeasured parts of our system and other performance inhibiting overheads.

Image size is influential for the performance of our tool when operating in server-mode. For our evaluations, the graph window was maintained 600x600 pixels to fit in the browser at most screen resolutions, producing for example, an average image size of 34kB for $G5$. However, since we are interested in potentially huge graphs, which may require more screen estate to display adequately, we also considered the impact of bigger window sizes. When we increase the window size to 1200x1200 pixels (4 times the original area), the average image size becomes 154kB. Running the system in server-mode with 600x600 pixel screen size takes 648 ms per frame while the 1200x1200 size takes 1051 ms per frame over a 1000 kB/s network. This is due to network overhead, and increased rendering times since graphical primitives contain more pixels. To summarize, most of the delay in our web-based graph visualization framework can be attributed to rendering while other delays account for only about 20% of the total.

Graph	*G1*	*G2*	*G3*	*G4*	*G5*	*G6*
Mode	Client	Client	Server	Server	Server	Server
Rendering	89%	83%	39%	57%	71%	77%
Interaction	0.04%	0.04%	0.4%	3.9%	6.5%	7.1%
Network	0%	0%	59%	30%	5%	0.5%
Other	11%	17%	2%	9%	18%	16%
Total *ms*	13.2	40.9	57.9	108.6	705.6	6299.5

Table 3: Percentage breakdown of the online interactive visualization process in Google Chrome for our test graphs.

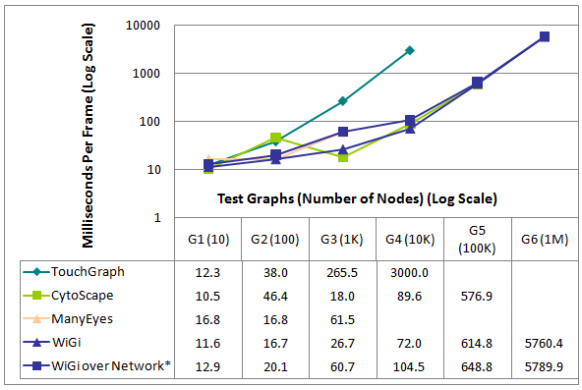
4.7 Comparison

To conclude the evaluation of our system, we now discuss a comparative test against three popular graph visualization systems: Touchgraph Navigator [17] (A Java Applet), IBM Many Eyes [6] (Java Applet), and Cytoscape [14] (a desktop application). A direct comparison with the plug-in free web-based version of Tom Sawyer Visualization was desired, but since this discussion focuses on scalable interaction, a direct comparison became infeasible because we were unable to interact with graphs in that system when more than a few hundred nodes are displayed. Our test dataset from Table 1 was converted to appropriate formats for each system and interactions timings were recorded for each using FRAPS, while keeping all graph elements in the viewing window. We note that the primary focus for these applications is not necessarily on scalability as they have many rich data exploration features for a variety of specific tasks, but this experiment does highlight that some of these systems are quite limited in scale. Since the other systems did not support interaction with the full graph based on single node movements, we restricted our system to movement of one node only. However, we note that in the worst-case, when the entire graph is repositioned based on the interaction algorithm, the timings for *WiGis* increase only by a very small amount. Since our system runs natively in the browser, FRAPS could not record timings. A JavaScript test harness was written to emulate a real user interacting with the graph. (Note: manual tests were also performed and similar results were achieved.) A click was simulated on a random node and it was moved to a random position in the view window, thus triggering selection and movement processes. The movement step was repeated 500 times and an average time was recorded. The experiment was repeated for graphs $G1, ..., G6$. Our system was tested with the browser running on the same machine as the server, and then network overhead with a connection of 1000kB/s was projected based on the image sizes. The fastest mode was used, which was server-side for all except when network overhead was included on $G1$ and $G2$.

Figure 6 shows the results of the interaction experiment in Google Chrome. For graphs of size $G3$ or less, all the systems completed the test in less than 100 ms per frame on average, except Touchgraph which

took 265 ms per frame. Our system showed a time increase with respect to graph size that is slightly less than linear. This occurs because overheads such as network time take up a smaller percentage of the overall process as graph size increases. The best performer from the other systems was Cytoscape, which took 570 ms for *G*5, which was 37.9 ms (6%) faster than our tool. An interesting trend in the graph occurs between *G*2 and *G*3 on the Cytoscape plot, where time per frame is reduced by 28 ms despite the increase in graph size. This occurs because Cytoscape renders nodes as squares instead of circles for graphs above a certain size. *WiGis* completed the test for *G*6 in an average of 5.7 seconds. These results show that the server side technique used in our system is more efficient than current graph visualization standards on the web.

The blank spaces in the table of Figure 6 represent failed attempts to load data. Under the setup described in Section 4.1, the largest of our test graphs we could load in Many Eyes was *G*3. TouchGraph failed at *G*5, while Cytoscape failed at *G*6. *WiGis* was the only system to successfully load the million node graph *G*6. Furthermore, we were unable to find another web-based graph visualization tool that could display graphs of the order of *G*5 or higher.



	G1 (10)	G2 (100)	G3 (1K)	G4 (10K)	G5 (100K)	G6 (1M)
TouchGraph	12.3	38.0	265.5	3000.0		
CytoScape	10.5	46.4	18.0	89.6	576.9	
ManyEyes	16.8	16.8	61.5			
WiGi	11.6	16.7	26.7	72.0	614.8	5760.4
WiGi over Network*	12.9	20.1	60.7	104.5	648.8	5789.9

Figure 6: Results of scalability and speed comparison against other systems. *Network delay over a 1000 kB/s network connection is estimated based on average image size.

Load times for each system were also noted, as they contribute greatly to the overall user experience. *WiGis* outperformed all other systems for every graph. *G*1 and *G*2 were loaded by all systems in less than 1 second. *WiGis*, Touchgraph and Cytoscape loaded *G*3 in less than one second, while Many Eyes took 5 s. Only *WiGis* and Cytoscape loaded *G*5, taking 2.6 and 4 seconds respectively, making *WiGis* 1.5 times faster.

5 Discussion And Conclusion

The main contribution of this paper to the graph drawing community is a framework for interactive visualization of large graphs over the web. We have presented an argument for our choice of a native browser implementation over a plug-in based approach. The framework provides user interaction with hundreds of thousands of nodes through the use of bitmap graph representations streamed from a remote server. Another novel contribution is the automatic switching between client and server graph models to maximize use of available resources. This is done in a manner which is transparent to the end user.

The approach used in the *WiGis* framework has several limitations. Firstly, since we are transferring data across a network there is a potential security risk and potential for data-loss. This can be mitigated somewhat by the use of SSL communications. Secondly, since we have chosen to display graphs natively in the browser, the current implementation cannot make use of rich functionality provided by plug-ins such as Java and Flash. Thirdly, as shown in our evaluations, the biggest bottleneck in our system occurs during rendering. There are a few possible avenues to address this issue. For example, use of a more powerful rendering technique, such as GPU rendering. Another possible improvement is to keep track of the nodes that will be re-rendered each frame and render those on top of an image of all the static nodes.

We have presented a detailed breakdown of the various components of the system in terms of speed and scalability. We compared *WiGis* against three popular systems and showed that our framework outperforms the best performing web-based system we could find by an order of magnitude in terms of scalability and achieves similar scale to the best performing desktop-based systems. In addition to the scalability advantages of our system, the fact that it is fully web-based (i.e: native) gives it the flexibility and ease-of-use to easily be applied to solve real-world graph visualization problems where users need to access graph data quickly and easily. For example, the tool is currently deployed by the U.S government in Blackbook- a data integration and search system used for counter-terrorism [11]. In this tool, *WiGis* visualize interconnections

between artifacts from a variety of diverse datasets, such as security reports or financial information. At the University of California, Irvine, *WiGis* have been deployed for visualization of a topic detection system [5] for newspaper articles. With a view to gathering useful and informative feedback on our visualization and interaction techniques from a large number of users, we are currently deploying a *WiGis* application on Facebook to visualize networks of friends and their various tastes in music, movies, etc. Due to the flexible nature of the framework it is easily adaptable to this task and we hope to report results of user evaluations in a future publication.

6 Acknowledgements

This work was partially supported by NSF grant IIS-0635492 through funds from the ITIC/IARPA KDD program, by NSF grants CNS-0722075 and IIS-0808772, as well as an ARO MURI award for proposal #56142-CS-MUR.

References

[1] James Abello and Jeffrey Korn. Mgv: A system for visualizing massive multi-digraphs. *IEEE Transactions on Visualization and Computer Graphics*, 8:21–38, 2002.

[2] D. Auber. Tulip. In P. Mutzel, M. Jünger, and S. Leipert, editors, *9th Symp. Graph Drawing*, volume 2265 of *Lecture Notes in Computer Science*, pages 335–337. Springer-Verlag, 2001.

[3] Albert-Laszlo Barabasi and Reka Albert. Emergence of scaling in random networks. *Science*, 286:509, 1999.

[4] Vladimir Batagelj and Andrej Mrvar. Pajek - program for large network analysis. *Connections*, 21:47–57, 1998.

[5] Chaitanya Chemudugunta, Padhraic Smyth, and Mark Steyvers. Text modeling using unsupervised topic models and concept hierarchies. *CoRR*, abs/0808.0973, 2008.

[6] Catalina M. Danis, Fernanda B. Viegas, Martin Wattenberg, and Jesse Kriss. Your place or mine?: visualization as a community component. In *CHI '08: Proceeding of the twenty-sixth annual SIGCHI conference on Human factors in computing systems*, pages 275–284, New York, NY, USA, 2008. ACM.

[7] P. Eades. A heuristic for graph drawing. *Congressus Numerantium*, 42:149–160, 1984.

[8] P. Eades and M. Huang. Navigating clustered graphs using force-directed methods, 2000.

[9] Thomas M. J. Fruchterman and Edward M. Reingold. Graph drawing by force-directed placement. *Softw. Pract. Exper.*, 21(11):1129–1164, 1991.

[10] Herman, G. Melan, and M. S. Marshall. Graph visualization and navigation in information visualization: A survey. *IEEE Transactions on Visualization and Computer Graphics*, 6(1):24–43, /2000.

[11] Intelligence Technology Innovation Center (ITIC). Blackbook prototype framework for the knowledge discovery and dissemination (kdd) program. McLean, VA, USA, October 3–4 2006.

[12] S. Milgram. The small world problem. *Psychology Today*, (2):60–67, 1967.

[13] Bruno Pinaud, Pascale Kuntz, and Fabien Picarougne. The website for graph visualization software references (gvsr). In *Graph Drawing*, pages 440–441, 2006. available at http://gvsr.polytech.univ-nantes.fr/GVSR/.

[14] P. Shannon, A. Markiel, O. Ozier, N. S. Baliga, J. T. Wang, D. Ramage, N. Amin, B. Schwikowski, and T. Ideker. Cytoscape: a software environment for integrated models of biomolecular interaction networks. *Genome Res*, 13(11):2498–2504, November 2003.

[15] Ben Shneiderman. The eyes have it: A task by data type taxonomy for information visualizations, 1996.

[16] Tom Sawyer Software. Tom sawyer visualization, 2009.

[17] Touchgraph. Touchgraph navigator. Proprietary online application, Touchgraph inc. available at http://www.touchgraph.com.

[18] Peterson Trethewey and Tobias Höllerer. Interactive manipulation of large graph layouts. Technical report, Department of Computer Science, University of California, Santa Barbara., 2009.

[19] D. J. Watts and S. H. Strogatz. Collective dynamics of 'small-world' networks. *Nature*, (393):440–442, 1998.

[20] Jason Wood, Ken Brodlie, and Helen Wright. Visualization over the world wide web and its application to environmental data. In *VIS '96: Proceedings of the 7th conference on Visualization '96*, pages 81–ff., Los Alamitos, CA, USA, 1996. IEEE Computer Society Press.